Mohammad Fadhil Rasyidin Parinduri // 5025201131
DAA – C
Misbakhul Munir Irfan Subakti, S.Kom., M.Sc.

FINAL EXAM

1. **True (T) or False (F) questions**
    1) Quick-sort has worse asymptotic complexity than merge-sort **(T)**
    2) Binary search is O(log n) **(T)**
    3) Linear-search in an unsorted array is O(n), but linear-search in a sorted array is O(log n) **(T)**
    4) Linear-search in a sorted linked list is O(n) **(T)**
    5) If you are only going to look up one value in an array, asymptotic complexity favours doing linear-search on the unsorted array over sorting the array and then doing a binary search **(T)**
    6) If all arc weights are unique, the minimum spanning tree of a graph is unique **(T)**
    7) Binary search in an array requires that the array is sorted **(T)**
    8) Insertion into an ordered list can be done in O(log n) time **(F)**
    9) A good hash function is one which tends to distribute elements uniformly throughout the hash table **(T)**
    10) In practice, with a good hash function and non-pathological data, objects can be found in O(1) time if the hash table is large enough **(T)**
    11) If a piece of code has asymptotic complexity O(g(n)), then at least g(n) operations will be executed whenever the code is run with parameter n **(F)**
    12) Given good implementations for different algorithms for some process such as sorting, searching or finding a minimum spanning tree, you should always choose the algorithm with the better asymptotic complexity **(F)**
    13) It is not possible for the depth-first and breadth-first traversal of a graph to visit nodes in the same sequence if the graph contains more than two nodes **(F)**
    14) The maximum number of nodes in a binary tree of height H (H = 0 for leaf nodes) is 2H+1 – 1 **(T)**
    15) In a complete binary tree, only leaf nodes have no children **(T)**
2. Please write down a polynomial-time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers. Please explain in detail your assumption, the pseudo-code of this algorithm, and explain the time complexity of this algorithm. (Assume that each integer appears once in the input sequence of n numbers)

    Assumption:
    1) Let arr[0], arr[1], arr[2], …, arr[n] be the input of sequence
    2) Let L[n] be array to store sub-problem solution. L[i] will store the length of the longest increasing subsequence ends with arr[i]
    3) Longest increasing subsequence ending with arr[0] has length 1

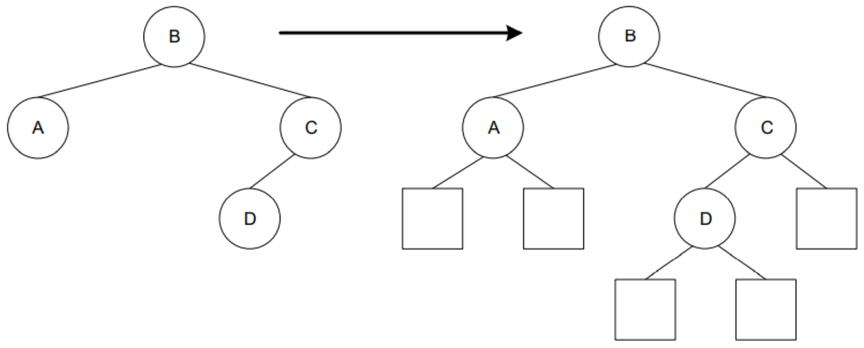
    Pseudo-code:

    We will use iterative function and below is the pseudo-code for the function

```
L[0] = 1

for i = 1 to n do

    L[i] = 1
```

```
        for j = 0 to i do
              if arr[i] > arr[j] and L[i] < L[j] + 1 then
                    L[i] = L[j] + 1;
                    endif
              endfor
        endfor
        return the maximum value in L[]
```

3. After all, binary trees have been discussed a lot in our lecture. It's often used in search applications, where the tree is searched by starting at the root and then proceeding either to the left or the right child, depending on some condition. In some instances, the search stops at a node in the tree. However, in some cases, it attempts to cross a null link to a non-existent child. The search is said to "fall out" of the tree. The problem with falling out of a tree is that you have no information about where this happened, and often this knowledge is useful information. Given any binary tree, an extended binary tree is formed by replacing each missing child (a null pointer) with a special leaf node, namely an external node. The remaining nodes are called internal nodes. An example is shown in the figure below, where the external nodes are shown as squares and the internal nodes are shown as circles. Note that if the original tree is empty, the extended tree consists of a single external node. Also, observe that each internal node has exactly two children and each external node has no children. Let n denote the number of internal nodes in an extended binary tree. An extended binary tree with n internal nodes has n + 1 external node. Prove this statement by induction.



Strong Induction

Strong (or course-of-values) induction is an easier proof technique than ordinary induction because you get to make a stronger assumption in the inductive step. In that step, you are to prove that the proposition holds for k+1 assuming that that it holds for all numbers from 0 up to k. This stronger assumption is especially useful for showing that many recursive algorithms work. The recipe for strong induction is as follows:

1) State the proposition P(n) that you are trying to prove to be true for all n.
2) Base case: Prove that the proposition holds for $n = 0$, i.e., prove that P(0) is true.
3) Inductive step: Assuming the induction hypothesis that P(n) holds for all n between 0 and k, prove that P(k+1) is true.
4) Conclude by strong induction that P(n) holds for all $n \geq 0$.

For example, consider a binary search algorithm that searches efficiently for an element contained in a sorted array. We might implement this algorithm recursively as follows:

```
/** Return an index of x in a.
 *  Requires: a is sorted in ascending order, and x is found in the
array a
 *  somewhere between indices left and right.
 */
int binsrch(int x, int[] a, int left, int right) {
    int m = (left+right)/2;
    if (x == a[m]) return m;
    if (x < a[m])
      return find(x, a, l, m−1)
    else
      return find(x, a, m+1, r);
}
```

Because this code is tail-recursive, we can also transform it into iterative code straightforwardly:

```
int binsrch(int x, int[] a, int left, int right) {
  while (true) {
    int m = (left+right)/2;
    if (x == a[m]) return m;
    if (x < a[m])
      r = m−1;
    else
      l = m+1;
  }
}
```

Binary search is efficient and easy to understand, but it is also famously easy to implement incorrectly. So it is a good example of code for which we want to think carefully about whether it works. Just testing it may well miss cases in which it does not work correctly. We can prove either piece of code correct by induction, but it is arguably simpler to think about the recursive version. The problem with convincing ourselves that binsrch works is that it uses itself, so the argument becomes circular if we're not careful. The key observation is that binsrch works in a divide-and-conquer fashion, calling itself only on arguments that are “smaller” in some way. In what way do the arguments become smaller? The difference between the parameters right and left becomes smaller in the recursive call. This is then the variable we should choose to construct our inductive argument. Now we can follow the strong induction recipe.

1) Let P(n) be the assertion that binsrch works correctly for inputs where right−left = n. If we can prove that P(n) is true for all n, then we know that binsrch works on all possible arguments.
2) Base Case. In the case where n=0, we know left=right=m. Since we assumed that the function would only be called when x is found between left and right, it must be the case that x = a[m], and therefore the function will return m, an index of x in array a.

3) Inductive Step. We assume that binsrch works as long as left−right ≤ k. Our goal is to prove that it works on an input where left−right = k + 1. There are three cases, where x = a[m], where x < a[m] and where x > a[m].
    - Case x = a[m]. Clearly the function works correctly.
    - Case x < a[m]. We know because the array is sorted that x must be found between a[left] and a[m−-1]. So if the recursive call works correctly, this call will too. The n for the recursive call is n = m − 1 − left = ⌊(left+right)/2⌋ − 1 − left. (Note that ⌊x⌋ is the floor of x, which rounds it down toward negative infinity.) If left+right is odd, then n = (left + right - 1)/2 − 1 − left = (right − left)/2 − 1, which is definitely smaller than right−left. If left+right is even then n = (left + right)/2 − 1 − left = (right−left)/2, which is also smaller than k + 1 = right−left because right−left = k + 1 > 0. So the recursive call must be to a range of a that is between 0 and k cells, and must be correct by our induction hypothesis.
    - Case x > a[m]. This is more or less symmetrical to the previous case. We need to show that r − (m + 1) ≤ right − left. We have r − (m + 1) − l = right − ⌊(left + right)/2⌋ − 1. If right+left is even, this is (right−left)/2 − 1, which is less than right−left. If right+left is odd, this is right− (left + right − 1)/2 − 1 = (right−left)/2 − 1/2, which is also less than right−left. Therefore, the recursive call is to a smaller range of the array and can be assumed to work correctly by the induction hypothesis.
4) Because in all cases the inductive step works, we can conclude that binsrch (and its iterative variant) are correct!

Notice that if we had made a mistake coding the x > a[m] case, and passed m as left instead of m+1 (easy to do!), the proof we just constructed would have failed in that case. And in fact, the algorithm could go into an infinite loop when right = left + 1. This shows the value of careful inductive reasoning. Showing Binary Search correct using induction (cornell.edu) **(source)**

Conclusion

Claim: A binary tree with n nodes has n + 1 null child links.

Proof: (by induction on the size of the tree) Let x(n) denote the number of null child links in a binary tree of n nodes. We want to show that for all n > 0, x(n) = n + 1. We'll make the convention of defining x(0) = 1. (If you like, you can think of the null pointer to
the root node as this link.)
For the basis case, n = 0, by our convention x(0) = 1, which satisfies the desired relation. For the induction step, let's assume that n ≥ 1. The induction hypothesis states that, for all n' < n, x(n') = n' +1. A binary tree with at least one node consists of a root node and two (possibly empty) subtrees, TL and TR. Let nL and nR denote the numbers of nodes in the left and right subtrees, respectively. Together with the root, these must sum to n, so we have n = 1 + nL + nR. By the induction hypothesis, the numbers of null links in the left and right subtrees are x(nL) = nL +1 and x(nR) = nR + 1. Together, these constitute all the null links. Thus, the total number of nul1 links is

x(n) = x(nL) + x(nR) = (nL + 1) + (nR + 1) = (1 + nL + nR) + 1 = n+ 1,

as desired.

4. **(1)** Triangle-Inequality:

The least distant path to reach a vertex j from i is always to reach j directly from i, rather than through some other vertex k (or vertices), i.e., dis(i, j) is always less than or equal to dist(i, k) + dist(k, j). The Triangle-Inequality holds in many practical situations. When the cost function satisfies the triangle inequality, it can be designed into an approximate algorithm for TSP that returns a tour whose cost is never more than twice the cost of an optimal tour. The idea is to use Minimum Spanning Tree (MST).

Algorithm:

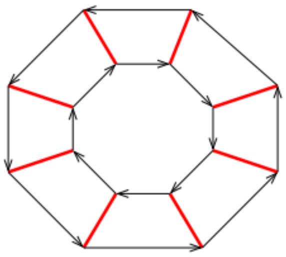
i. Let 1 be the starting and ending point for salesman.
ii. Construct MST from with 1 as root using Prim's Algorithm.
iii. List vertices visited in pre-order walk of the constructed MST and add 1 at the end.

(2) Running time:

The running time is polynomial because it's using Prim's Algorithm which has O(V^2) running time

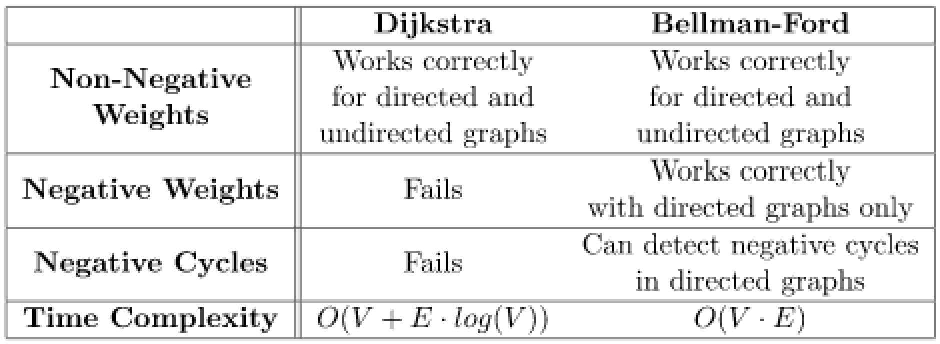
(3) ratio bound (P)

In figure below, a directed graph is shown for which the integrality gap is 3/2. Each directed edge has LP value 1/2. The total LP value is therefore 2 · 2k · 1/2 for the black edges plus 2k for the red edges for a total LP value of 4k. Meanwhile, the optimal integer solution is ≈ 6k. It has been conjectured that the integrality gap of both P_{ATSP}^{met} and P_{ATSP} is at most 2.



Each inner and outer directed cycle consists of k black edges. Each bold red edge denotes a double directed path of length two. Black edges have weight 2 and red edges have weight 1.

5. **(1)** Look at the similarities and differences between Dijkstra's and Bellman-Ford algorithms:



	Dijkstra	**Bellman-Ford**
Non-Negative Weights	Works correctly for directed and undirected graphs	Works correctly for directed and undirected graphs
Negative Weights	Fails	Works correctly with directed graphs only
Negative Cycles	Fails	Can detect negative cycles in directed graphs
Time Complexity	$O(V + E \cdot log(V))$	$O(V \cdot E)$

As we can see, Dijkstra’s algorithm is better when it comes to reducing the time complexity. However, when we have negative weights, we have to go with the Bellman-Ford algorithm. Also, if we want to know whether the graph contains negative cycles or not, the Bellman-Ford algorithm can help us with that.

Just one thing to remember, in case of negative weights or even negative cycles, the Bellman-Ford algorithm can only help us with directed graphs.

Dijkstra -- The main advantage of Dijkstra’s algorithm is its considerably low complexity, which is almost linear. However, when working with negative weights, Dijkstra’s algorithm can’t be used. Also, when working with dense graphs, where E is close to V^2, if we need to calculate the shortest path between any pair of nodes, using Dijkstra’s algorithm is not a good option.

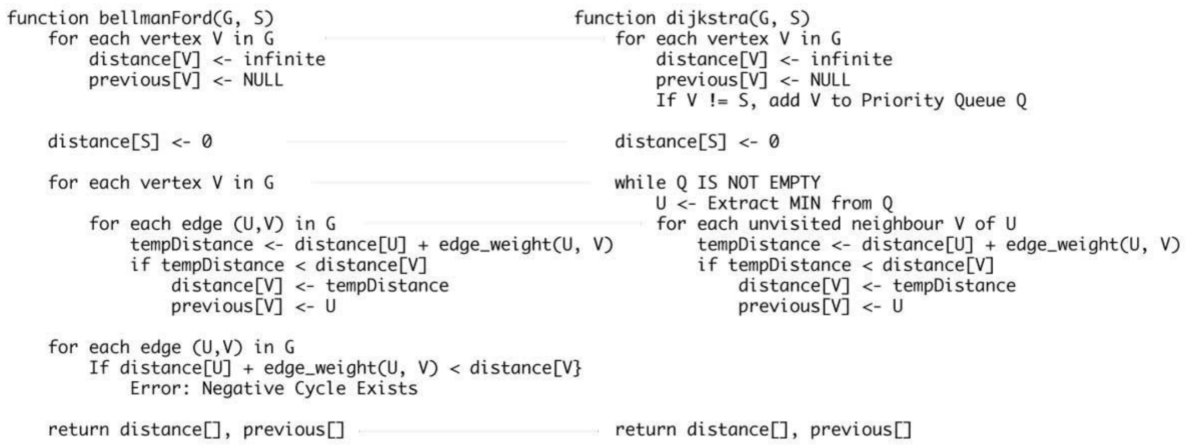
The reason for this is that Dijkstra’s time complexity is O(V + E \cdot log(V)). Since E equals almost V^2, the complexity becomes O(V + V^2 \cdot log(V)). When we need to calculate the shortest path between every pair of nodes, we’ll need to call Dijkstra’s algorithm, starting from each node inside the graph. Therefore, the total complexity will become O(V^2 + V^3 \cdot log(V)).

The reason why this is not a good enough complexity is that the same can be calculated using the Floyd-Warshall algorithm, which has a time complexity of O(V^3). Hence, it can give the same result with lower complexity.

Bellman-Ford -- The main advantage of the Bellman-Ford algorithm is its capability to handle negative weights. However, the Bellman-Ford algorithm has a considerably larger complexity than Dijkstra’s algorithm. Therefore, Dijkstra’s algorithm has more applications, because graphs with negative weights are usually considered a rare case.

As mentioned earlier, the Bellman-Ford algorithm can handle directed and undirected graphs with non-negative weights. However, it can only handle directed graphs with negative weights, as long as we don’t have negative cycles. Also, we can use the Bellman-Ford algorithm to check the existence of negative cycles, as already mentioned.

(2)



```
function bellmanFord(G, S)
    for each vertex V in G
        distance[V] <- infinite
        previous[V] <- NULL

    distance[S] <- 0

    for each vertex V in G

        for each edge (U,V) in G
            tempDistance <- distance[U] + edge_weight(U, V)
            if tempDistance < distance[V]
                distance[V] <- tempDistance
                previous[V] <- U

    for each edge (U,V) in G
        If distance[U] + edge_weight(U, V) < distance[V}
            Error: Negative Cycle Exists

    return distance[], previous[]
```

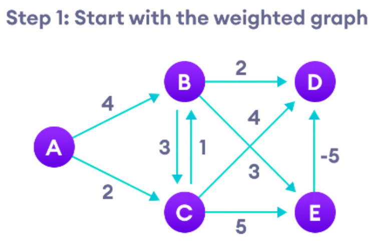
```
function dijkstra(G, S)
    for each vertex V in G
        distance[V] <- infinite
        previous[V] <- NULL
        If V != S, add V to Priority Queue Q

    distance[S] <- 0

    while Q IS NOT EMPTY
        U <- Extract MIN from Q
        for each unvisited neighbour V of U
            tempDistance <- distance[U] + edge_weight(U, V)
            if tempDistance < distance[V]
                distance[V] <- tempDistance
                previous[V] <- U

    return distance[], previous[]
```

Bellman Ford's algorithm and Dijkstra's algorithm are very similar in structure. While Dijkstra looks only to the immediate neighbors of a vertex, Bellman goes through each edge in every iteration.

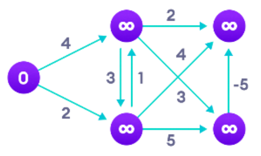
Bellman-Ford -- Needs to maintain the path distance of every vertex. We can store that in an array of size v, where v is the number of vertices. We also want to be able to get the shortest path, not only know the length of the shortest path. For this, we map each vertex to the vertex that last updated its path length. Once the algorithm is over, we can backtrack from the destination vertex to the source vertex to find the path.

Dijkstra -- Needs to maintain the path distance of every vertex. We can store that in an array of size v, where v is the number of vertices. We also want to be able to get the shortest path, not only know the length of the shortest path. For this, we map each vertex to the vertex that last updated its path length. Once the algorithm is over, we can backtrack from the destination vertex to the source vertex to find the path. A minimum priority queue can be used to efficiently receive the vertex with least path distance.

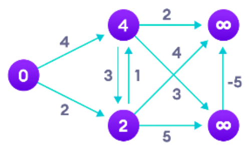
(3) **Bellman-Ford**



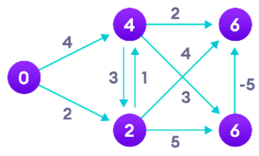
Step 2: Choose a starting vertex and assign infinity path values to all other vertices



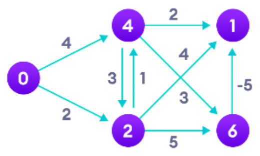
Step 3: Visit each edge and relax the path distances if they are inaccurate



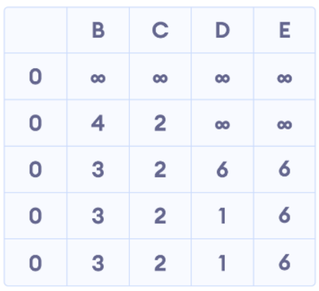
Step 4: We need to do this V times because in the worst case, a vertex's path length might need to be readjusted V times



Step 5: Notice how the vertex at the top right corner had its path length adjusted

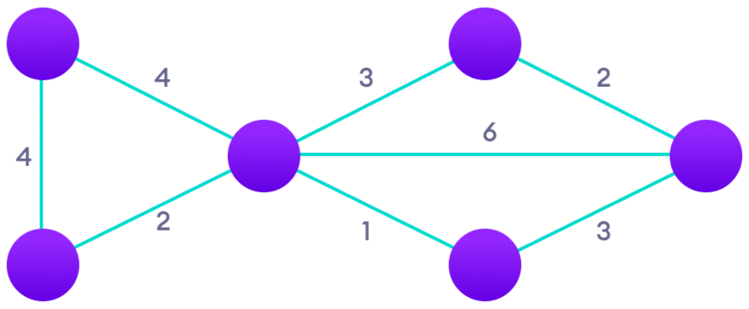


Step 6: After all the vertices have their path lengths, we check if a negative cycle is present



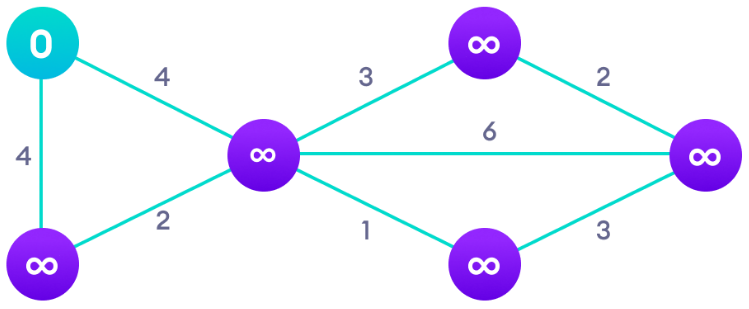
	B	C	D	E
0	∞	∞	∞	∞
0	4	2	∞	∞
0	3	2	6	6
0	3	2	1	6
0	3	2	1	6

Dijkstra



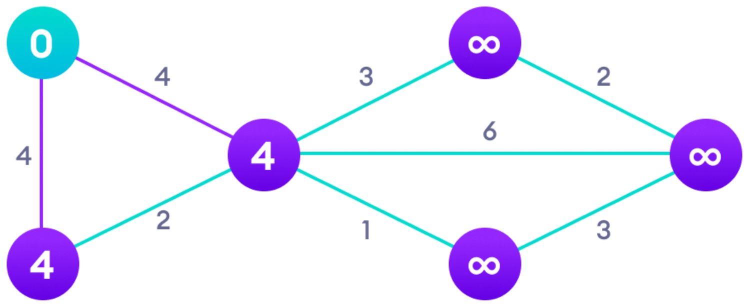
Step: 1

Start with a weighted graph



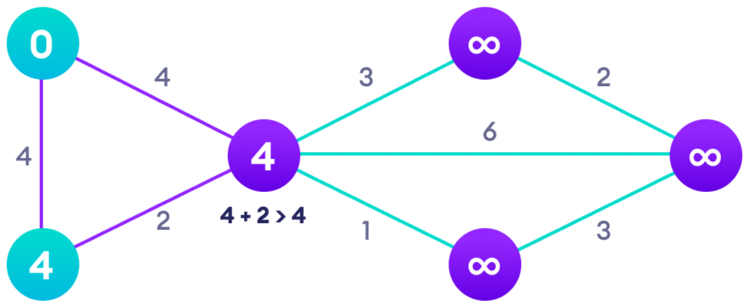
Step: 2

Choose a starting vertex and assign infinity path values to all other devices



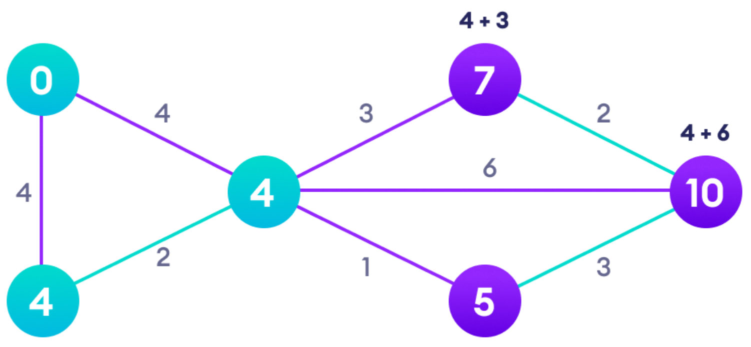
Step: 3

Go to each vertex and update its path length



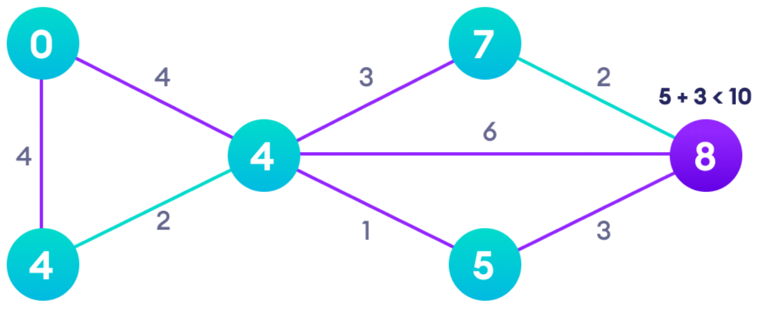
Step: 4

If the path length of the adjacent vertex is lesser than new path length, don't update it



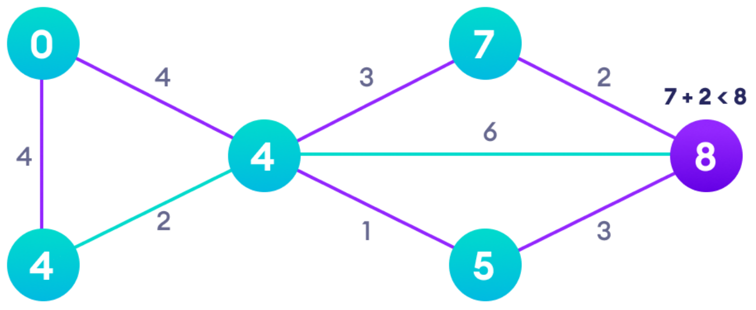
Step: 5

Avoid updating path lengths of already visited vertices



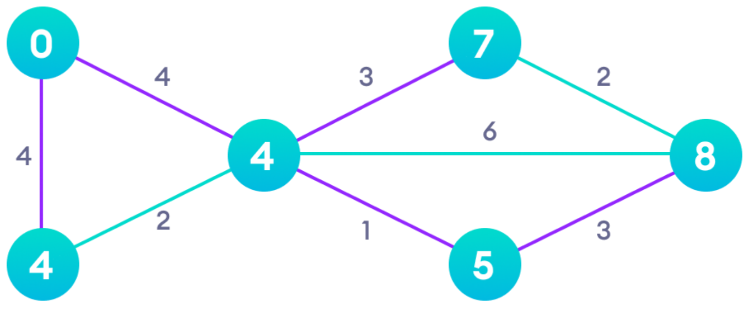
Step: 6

After each iteration, we pick the unvisited vertex with the least path length. So we choose 5 before 7



Step: 7

Notice how the rightmost vertex has its path length updated twice



Step: 8

Repeat until all the vertices have been visited